6.2. The Management of Binding Environments

While this thesis concentrated on the discussion of controlling parallel activities in an execution model, another important problem, the management of binding environments, was given little attention. This problem is important since for effective computation, the variable bindings generated in a unification must be maintained if they are necessary for later unifications or the final answer constructions (see Section 1.2.4). In this section, a brief discussion of the problem is provided.

In a conventional Prolog implementation, there are basically two alternative approaches of representing the binding environments: *structure-sharing* [BoM72, Bru82] and *structure-copying* [Mel82]. In both approaches, the binding of a variable to a constant, another variable, or a ground complex term can be a pointer to the direct representation of the value. To represent the binding to a non-ground complex term, the structure-sharing approach uses two pointers: one to the pure code of the term, another to the binding environment containing the variables occur in the term; while the structure-copying approach creates a new copy of the term for the binding. A comparison of the two approaches, mainly on memory consumptions, is provided by Mellish [Mel82].

Additional problems reveal themselves when a parallel implementation is considered, though the basic ideas of the two above approaches can still be applied. A major problem is the need to manage separate OR-parallel binding environments, due to the mutually exclusive bindings made to the same goal variable by the OR-parallel computations. This problem is not serious for PARLOG and GHC, since such bindings are either disallowed or suspended before commitment, and only one of them can be made after commitment. However, the situation is quite complicated in Concurrent Prolog. The semantics of Concurrent Prolog allow bindings to goal variables to be made locally, but disallow them to be accessed outside the guard before commitment.

A natural approach to cope with this is to use the copying scheme, i.e. to make a copy of the goal for each unification so that goal variables are bound in the copied goal, not the original one. The copied and original goals are unified at commitment. However, because of the copying, the links for information flow from goals to guards are broken, creating unexpected results in guard evaluation that depend on this information. Two recently proposed schemes: ***multiple commitment*** and ***eager broadcasting*** [LeF86] are still not satisfactory; the former does not solve the problem, the latter solves the problem, but requires complex run-time structures and incurs high overheads. For parallel Prolog, various techniques such as the schemes of directory trees, hash windows, etc. are proposed to avoid unnecessary duplication of binding information; a brief survey of these schemes is provided by Levy and Friedmann ([LeF86], pages 19-24).

Another problem is the efficient support of binding environments when a distributed implementation is considered. The binding environments generated during computations are highly inter-connected, distributing them among different processing elements implies a complex variable reference scheme and additional message overheads in dereferencing a variable, which is a most basic operation in the system (e.g. Port Prolog [Lee84]). A possible solution to the above problem may be to completely ban the notion of binding environment, but literally substitute the bound variables in a goal with the terms (e.g. the AND/OR Process Model [Con83], and PIE [GTM84]). No dereference of variable is needed in this scheme, but the copying overheads may be unacceptably high, especially when there is a large amount of structure data. More research is needed in order to reach a satisfactory solution.

6.3. Future Work

As the work reported in this thesis represents only a preliminary step towards an actual implementation of a parallel logic programming system, many extensions and improvements are remained to be done. In this section, some suggestions for possible directions of future work are provided.

Before making any full-scale implementation of the execution model, a first step may be to obtain some performance measurements through simulations or pilot implementations to verify the design decisions. A possible direction is to implement the execution model on a uniprocessor running an operating system (e.g. Unix) where multiple processes are supported so that a multiprocessor architecture can be simulated. The major benefits of this approach are the ease of making architectural changes for experimental purposes, and the alleviation of the difficulties in porting the developed programs in the prototyping environment to the target system, due to their closeness. In a pilot implementation, unifications and binding environment management operations can be simulated; however, simulation parameters such as the average times of a unification step and each of those operations must be provided.

The performance measurements should provide information such as the effectiveness of the model's failure/abortion propagation scheme and the goal suspension/re-activation scheme, the ratio of lock operations to unifications performed by each processor, the ratio of processor utilization time to idle time, the growth rate of memory contentions with the increasing number of processors, and the effectiveness of the model in exploiting the parallelism in logic programs and supporting architectures (e.g. to produce plots of the speedups in running different programs on architectures with different number of processors). An analysis of this information can help to pinpoint the shortcomings of the model and make enhancements.

A second step may be to address the problems in managing the binding environ-

ments. Since the three target languages have different unification algorithms and different binding environment management requirements, a decision must be made to specifically support one of the languages first. A good starting point may be to support GHC or a simpler version of PARLOG (e.g. PARLOG without the sequential AND/OR operators) where the management of OR-parallel binding environments is not that complicated, when compared with Concurrent Prolog. Finally, to extend the implementation to a logic programming system, various problems, such as the design of the internal program codes, the device of a garbage collection scheme (on a multiprocessor system), and the provision of program development tools (e.g. debugging facilities), must be effectively solved.

References

[BaM73] G. Battani and H. Meloni, *Interpreteur du Language de Programmation Prolog*, Groupe d'Intelligence Artificielle, University d'Aix Marseille, Luminy, France, 1973.

[Bic84] L. Bic, Execution of Logic Programs on a Dataflow Architecture, *Proc. of the 11th Annual International Symposium on Computer Architecture*, Ann Arbor, Michigan, U.S.A., 1984, 290-296.

[Bis86] P. Bishop, *Fifth Generation Computers: Concepts, Implementations, and Uses*, Ellis Horwood Limited, 1986.

[Bow82] K. A. Bowen, Concurrent Execution of Logic, *Proc. of the First International Logic Programming Conference*, Marseille, France, September 1982, 26-30.

[BoM72] R. S. Boyer and J. S. Moore, The Sharing of Structure in Theorem Proving Programs, in *Machine Intelligence*, vol. 7, Bernard Meltzer and Donald Michie (ed.), Edinburgh University Press, 1972.

[Bru82] M. Bruynooghe, The Memory Management of Prolog Implementations, in *Logic Programming*, K. L. Clark and S.-A. Tarnlund (ed.), 1982, 83-98.

[BrP84] M. Bruynooghe and L. M. Pereira, Deduction Revision by Intelligent Backtracking, in *Implementations of Prolog*, J. A. Campbell (ed.), Ellis Horwood Limited, 1984, 194-215.

[Che84] M. H. M. Cheng, The Design and Implementation of the Waterloo Unix Prolog Environment, Master Thesis, Department of Computer Science, University of Waterloo, 1984.

[CiH84] A. Ciepielewski and S. Haridi, Control of Activities in the OR-Parallel Token Machine, *Proc. of IEEE International Symposium on Logic Programming*, Atlantic City, NJ, U.S.A., 1984, 49-57.

[Cla78] K. L. Clark, Negation as Failure, in *Logic and Data Bases*, Herve Gallaire and Jack Minker (ed.), Plenum Press, New York, 1978, 293-322.

[ClG81] K. L. Clark and S. Gregory, A Relational Language for Parallel Programming, *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, October 1981, 171-178.

[CMG82] K. L. Clark, F. G. McCabe and S. Gregory, IC-PROLOG Language Features, in *Logic Programming*, K. L. Clark and S.-A. Tarnlund (ed.), 1982, 253-266.

[ClT82] K. L. Clark and S.-A. Tarnlund, *editors, Logic Programming*, Academic Press, 1982.

[ClG84] K. Clark and S. Gregory, Notes on Systems Programming in PARLOG, *Proc. of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, 299-306.

[ClM81] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.

[CoK81] J. S. Conery and D. F. Kibler, Parallel Interpretation of Logic Programs, *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, October 1981, 163-170.

[Con83] J. S. Conery, The AND/OR Process Model for Parallel Interpretation of Logic Programs, Technical Report 204, Ph.D. Thesis, University of California at Irvine, June 1983.

[Cox84] P. T. Cox, Finding Backtrack Points for Intelligent Backtracking, in *Implementations of Prolog*, J. A. Campbell (ed.), Ellis Horwood Limited, 1984, 216-233.

[DaR81] J. Darlington and M. Reeve, Alice: A Multi-processor Reduction Machine for the Parallel Evaluation of Applicative Languages, *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, 1981, 65-75.

[DeG84] D. DeGroot, Restricted And-Parallelism, *Proc. of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, 471-478.

[Fer81] R. J. Ferguson, Prolog Interpreter for the Unix Operating System, Master Thesis, Department of Computer Science, University of Waterloo, 1981.

[GTM84] A. Goto, H. Tanaka and T. Moto-oka, Highly Parallel Inference Engine PIE - Goal Rewriting Model and Machine Architecture, *New Generation Computing 2*, 1 (1984), 37-58.

[Gre85] S. Gregory, *Design, Application and Implementation of a Parallel Logic Programming Language*, Ph.D. Thesis, Imperial College of Science and Technology, University of London, September 1985.

[Hal86] Z. Halim, A Data-Driven Machine for OR-Parallel Evaluation of Logic Programs, *New Generation Computing 4*, 1 (1986), 5-33.

[ISK85] N. Ito, H. Shimizu, M. Kishi, E. Kuno and K. Rokusawa, Data-flow Based Execution Mechanisms of Parallel and Concurrent Prolog, *New Generation Computing 3*, 1 (1985), 15-41.

[KKM83] S. Kasif, M. Kohli and J. Minker, PRISM: A Parallel Inference System for Problem Solving, *Proc. of Logic Programming Workshop 83*, Portugal, 1983, 123-152.

[KKI85] M. Kishi, E. Kuno, N. Ito and K. Rokusawa, The Dataflow-Based Parallel Inference Machine to Support Two Basic Languages in KL1, Technical Report TR-114, ICOT, Tokyo, Japan, 1985.

[KoK71] R. A. Kowalski and D. Kuehner, Linear Resolution with Selection Function, *Artificial Intelligence 2*, (1971), 227-260.

[Kow74] R. A. Kowalski, Predicate Logic as Programming Language, *Proceedings of IFIP Congress 74*, Stockholm, Sweden, August 1974, 569-574.

[Kow79a] R. A. Kowalski, *Logic for Problem Solving*, Elsevier, North-Holland, 1979.

[Kow79b] R. A. Kowalski, Algorithm = Logic + Control, *Comm. ACM 22*, 7 (July 1979), 424-436.

[KuL86] V. Kumar and Y. J. Lin, *A Framework for Intelligent Backtracking in Logic Programs*, Computer Science Department, University of Texas at Austin, 1986.

[Lee84] K. S. Lee, Concurrent Prolog in a Multi-process Environment, CS-84-46, Department of Computer Science, University of Waterloo, November 1984.

[Lev84] J. Levy, A Unification Algorithm for Concurrent Prolog, *Proc. of the Second International Logic Programming Conference*, Uppsala, July 1984, 333-341.

[Lev86] J. Levy, Shared Memory Execution of Committed-Choice Languages, CS86-29, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, October 1986.

[LeF86] J. Levy and N. Friedmann, Concurrent Prolog Implementations - Two New Schemes, CS86-13, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, May 1986.

[Llo84] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1984.

[Mai83] D. Maier, *The Theory of Relational Databases*, Computer Science Press, 1983.

[Mel82] C. S. Mellish, An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter, in *Logic Programming*, K. L. Clark and S.-A. Tarnlund (ed.), 1982, 99-106.

[Mie84] C. Mierowsky, Design and Implementation of Flat Concurrent Prolog, CS84-21, The Weizmann Institute of Science, Rehovot, Israel, 1984.

[MTA84] T. Moto-oka, H. Tanaka, H. Aida, K. Hirata and T. Maruyama, The Architecture of a Parallel Inference Engine - PIE, *Proc. of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, November 1984, 479-488.

[MoK85] T. Moto-oka and M. Kitsuregawa, *The Fifth Generation Computer: the Japanese Challenge*, John Wiley & Sons, 1985.

[Nil80] N. J. Nilsson, *Principles of Artificial Intelligence*, SRI International, 1980.

[OAS85] R. Onai, M. Aso, H. Shimizu, K. Masuda and A. Matsumoto, Architecture of a Reduction-Based Parallel Inference Machine: PIM-R, *New Generation Computing 3*, 2 (1985), 197-228.

[Pol81] G. H. Pollard, *Parallel Execution of Horn Clause Programs*, Ph.D. Thesis, University of London, Imperial College of Science & Technology, 1981.

[Rob65] J. A. Robinson, A Machine Oriented Logic Based on the Resolution Principle, *J. ACM 12*, 1 (1965), 23-44.

[ShT83] E. Y. Shapiro and A. Takeuchi, Object Oriented Programming in Concurrent Prolog, *New Generation Computing 1*, (1983), 25-48.

[Sha83] E. Y. Shapiro, A Subset of Concurrent Prolog and its Interpreter, CS83-06, The Weizmann Institute of Science, Rehovot, Israel, February 1983.

[TaK84] N. Tamura and Y. Kaneda, Implementing Parallel Prolog on a Multiprocessor Machine, *Proc. of IEEE International Symposium on Logic Programming*, Atlantic City, NJ, U.S.A., 1984, 42-48.

[TLM84] S. Taylor, A. Lowry, G. Q. Maguire Jr. and S. J. Stolfo, Logic Programming Using Parallel Associative Operations, *Proc. of IEEE International Symposium on Logic Programming*, Atlantic City, NJ, U.S.A., 1984, 58-68.

[TBH82] P. C. Treleaven, D. R. Brownbridge and R. P. Hopkins, Data-Driven and Demand-Driven Computer Architecture, *ACM Computing Surveys 14*, 1 (March 1982), 93-143.

[Ued85] K. Ueda, Guarded Horn Clauses, Technical Report TR-103, ICOT, Tokyo, Japan, 1985.

[UmT83] S. Umeyama and K. Tamura, A Parallel Execution Model of Logic Programs, *Proc. of the 10th Annual International Symposium on Computer Architecture*, Stockholm, Sweden, June 1983, 349-355.

[vaK76] M. H. van Emden and R. A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *J. ACM 23*, 4 (October 1976), 733-742.

[vad79] M. H. van Emden and G. J. de Lucena, Predicate Logic as a Language for Parallel Programming, CS-79-15, University of Waterloo, April 1979.

[van81] M. H. van Emden, An Interpreting Algorithm for Prolog Programs, CS-81-28, University of Waterloo, September 1981.

[War77] D. H. D. Warren, Implementing Prolog: Compiling Predicate Logic Programs, D.A.I. Research Report No. 39/40, University of Edinburgh, Edinburgh, Scotland, 1977.

[War80] D. H. D. Warren, Logic Programming and Compiler Writing, *Software-Practice and Experience 10*, (1980), 97-125.

[WAD84] D. S. Warren, M. Ahamad, S. K. Debray and L. V. Kale, Executing Distributed Prolog Programs on a Broadcast Network, *Proc. of IEEE International Symposium on Logic Programming*, Atlantic City, NJ, U.S.A., 1984, 12-21.

[Wis82] M. J. Wise, A Parallel Prolog: the Construction of a Data Driven Model, *1982 ACM Symposium on Lisp and Functional Programming*, August 1982, 56-66.

Appendix A1

Example Programs

In this appendix, example programs in Concurrent Prolog, PARLOG and GHC are provided with the intention to give the reader some ideas of the programming techniques supported by these languages. The programs are very simple and their executions incur no guard hierarchies; more programming examples can be found in the references quoted in Section 2.2.

The first example is a stack process written in Concurrent Prolog as shown in Program 1:

```
[C0] stack(S) <- stack(S?, []).
[C1] stack([pop(X)|S], [X|Xs]) <- stack(S?, Xs).
[C2] stack([push(X)|S], Xs) <- stack(S?, [X|Xs]).
[C3] stack([], []).
```

Program 1: A Stack Process Written in Concurrent Prolog

A stack process (implemented by the clauses C1 to C3) has two arguments: the first is an input stream of stack operations represented as a list of commands, the second is the stack represented as a list of elements. The clause C0 is used as an interface to the caller; its purpose is to initialize the stack whose internal representation (i.e. a list) is hidden from the caller.

A stack process is invoked by a goal *stack(S?)*, where S is the input stream of stack operations. The goal unifies with the clause head of C0, invoking the stack process with an empty stack. The invoked stack process then iterates, processing the commands in the input stream. If no command is available (i.e., S is uninstantiated), the process suspends, due to the read-only annotation S? in the recursive calls to stack. Otherwise, one of the following three cases must apply:

(1) If the next command in the input stream is pop(X) and the stack is nonempty, the clause C1 is invoked. It unifies X with the top of the stack, and iterates with the rest of the input stream and the rest of the stack.

(2) If the next command in the input stream is push(X), the clause C2 is invoked. It adds X to the top of the stack, and iterates with the rest of the input stream and the new stack.

(3) If both the input stream and the stack are empty, the clause C3 is invoked to terminate the stack process.

If none of these cases applies (e.g. when the next command is pop(X) but the stack is empty), the stack process fails. Stream parallelism can be exploited if the input stream S is incrementally constructed by the caller, in that case the execution of the stack process is synchronized by the availability of data in S. In Concurrent Prolog, the absence of a guard from a clause means that the guard is true. Hence when the clauses C1 to C3 are tried in parallel, the first one with successful head unification will become the committed clause.

The second example is a simple Unix-like shell written in PARLOG. The shell, as shown in Program 2, handles a stream of commands to run foreground and background processes without input and output:

```
mode shell(?).

[C1] shell([]).
[C2] shell([bg(Proc)|Cmds]) <- call(Proc) , shell(Cmds).
[C3] shell([fg(Proc)|Cmds]) <- call(Proc) & shell(Cmds).
```

Program 2: A Simple Shell Written in PARLOG

The procedure shell is invoked by a goal *shell(Cmdlist)*, where Cmdlist is the input stream represented as a list of commands. Each command should be either bg(Proc) or fg(Proc), which denotes a background process or a foreground process, respectively. If no command is available (i.e. Cmdlist is uninstantiated), the clauses C1 to C3 become suspended since they violate the input constraint of the input argument, as indicated by the mode declaration shell(?). As a result, the goal will be suspended since all its applicable clauses are suspended. Otherwise, one of the following three cases must apply:

(1) If Cmdlist is empty, the clause C1 is used to terminate the shell.

(2) If the next command is bg(Proc), the clause C2 deals with this background command by concurrently (because of the parallel AND operator ",") invoking the process Proc via the metalevel facility call, and resuming the shell to process the next command.

(3) If the next command is fg(Proc), the clause C3 deals with this foreground command by invoking the process Proc via the metalevel facility call, and waiting for it to terminate successfully (because of the sequential AND operator "&") before resuming the shell to process the next command.

The goal fails if none of these cases applies. Stream parallelism can be exploited if Cmdlist is incrementally constructed by the caller. As in Concurrent Prolog, the absence of a guard from a clause means that the guard is true.

The example program for GHC is a binary merge process as shown in Program 3:

```
[C1] merge([A|Xs], Ys, Zs) <- true | Zs=[A|Zs1], merge(Xs, Ys, Zs1).
[C2] merge(Xs, [A|Ys], Zs) <- true | Zs=[A|Zs1], merge(Xs, Ys, Zs1).
[C3] merge([], Ys, Zs) <- true | Zs=Ys.
[C4] merge(Xs, [], Zs) <- true | Zs=Xs.
```

Program 3: A Binary Merge Process Written in GHC

The procedure merge is invoked by a goal *merge(Is1, Is2, Os)* which merges the two input streams Is1 and Is2 into the output stream Os. All the streams are implemented as lists of elements. If no data is available in both input streams (i.e., both Is1 and Is2 are uninstantiated), the clauses C1 to C4 become suspended since they violate the rule of suspension. As a result, the goal will be suspended since all its applicable

clauses are suspended. Otherwise, one of the following four cases must apply:

(1) If data in the first input stream is available, the clause C1 is invoked to add the data to the output stream, and iterates with the rest of the first input stream.

(2) If data in the second input stream is available, the clause C2 is invoked to add the data to the output stream, and iterates with the rest of the second input stream.

(3) If the first input stream is empty, the clause C3 is invoked to connect the second input stream to the output stream.

(4) If the second input stream is empty, the clause C4 is invoked to connect the first input stream to the output stream.

If data are available in both input streams, which of C1 and C2 is selected first depends on the particular implementation. In a good parallel implementation, the elements from the two input streams are expected to appear in the output stream almost in the order of arrival. Note that in GHC, a true guard must be explicitly stated. The binding of Zs to [A|Zs1] in C1 and C2 must be done explicitly in the clause body, instead of implicitly in the clause head, to avoid violating the rule of suspension.

Appendix A2
Levy's Execution Model

The proposed execution model described in Chapter 4 of this thesis is a modified version of Levy's execution model, which was designed to support the parallel execution of Concurrent Prolog, PARLOG and GHC on shared memory multiprocessor systems. Many of the design ideas in the proposed execution model are inherited from Levy's work; these include the viewing of the computation necessary to satisfy a goal with its applicable clauses as a unit, the encoding of the computations by a finite state machine, the heavy usage of locking operations and the concept of recursive commitment. As these features have already been covered in Chapter 4, to avoid repetition, this appendix will only give a brief description of Levy's execution model and highlight its major differences with the proposed execution model.

In Levy's execution model, the AND/OR tree of a logic program is partitioned into computation units each of which includes the computation necessary to satisfy a goal with all its applicable clauses. A computation unit is represented by a goal record and several guard tuples: the goal record represents the goal to be solved, the guard tuples represent the computations of individual applicable clauses (after their guards are spawned). Computation units are placed in the ready goal pool and are commonly accessed by a set of processes, each of which executes a copy of a finite-state machine that repeatedly selects a goal from the pool, advances its state, and places the generated goals (if any) back into the pool.

A goal record contains all the information necessary to satisfy a goal. It, as described in Levy's model, has the following fields: Procedure, Arguments, Guard_Tuple, Status_Vector, Env_Vector, Ready and Link. The first five fields are carried over to the proposed execution model; their meanings are described in Section 4.3.1. The Ready field is set to NULL when the goal record is not in the ready goal pool. It is set to some non-NULL value if it is currently in the ready goal pool. The Link field is the same as the R_Link field in the proposed execution model; it is used to manage the pool of ready goals.

Since Concurrent Prolog, PARLOG and GHC are committed choice languages, a goal will commit to one of its applicable clauses when the guard of that clause is solved, and the computations involving the other applicable clauses are useless after the commitment. Also, a guard is a conjunction; it fails when one of its goals fails. To avoid unnecessary computations, information of failure and abortion (due to commitment) should be propagated promptly: failure should be propagated across the conjunction of goals in a guard and up the AND-OR tree; abortion should be propagated across the disjunction of applicable clauses of a goal and down the AND-OR tree. In Levy's execution model, the goal records are not physically interconnected as a tree, so only the upward propagation of failure, by checking the Status_Vector field of a goal record, is supported. It is possible that Levy's model will waste its resources in the computations involving a subtree rooted by an aborted/failed goal. To avoid this situation, the proposed execution model adds two fields to the goal record which, together with the Father and Goal_List fields of the guard tuple (see below), reveal the AND-OR tree structure among the goal records along which failure and abortion can be propagated. These two fields are GT_Vector and G_Link, their meanings are described in Section 4.3.1.

Another modification in the proposed execution model is to replace the Ready field in the goal record by the Flag field. The Flag field is set to READY if the goal

record is currently in the ready goal pool. It is set to DEQUEUED if it is removed from the ready goal pool and is in some execution status. It is set to DEAD if it has failed, been reduced or aborted. This further classification of goal status is necessary since the computations involving a goal in the ready goal pool may become useless, say, because of the propagation of failure and abortion to that goal. The classification of goal records into READY and non-READY, as in Levy's execution model, is not adequate to reflect this fact.

A guard tuple is used to monitor the computation of a guard. It, as described in Levy's model, has three fields: Goal_Record, Status_Word, and Guard_Count. The Goal_Record field contains a reference to the goal for which this guard is computing, it is the same as the Father field in the proposed model. The Status_Word field points to the location of the status vector of the goal corresponding to the clause containing this guard. In the proposed model, it is replaced by a pointer to the clause so that the committed clause can be located in recursive commitment. The Guard_Count field records the number of currently unsolved goals in this guard; it is used in recursive commitment (see Section 4.4.2.5). A fourth field, Goal_List, is added to the guard tuple in the proposed model; it is used to manage the goals in this guard (see Section 4.3.3).

In Levy's model, a goal reduction step is divided into seven stages: UNIFY, UNIFYING, SPAWN, SPAWNED, COMMIT, COMMITTED and FAIL. This staged goal reduction concept is carried over to the proposed model, the meanings of the stages are described in Section 4.3.2. Because of the introduction of failure/abortion propagation in the proposed model, a new stage, KILLED, is added. When a goal commits to a clause, the status words of other applicable clauses of that goal are set to KILLED (in Levy's model the status words are set to FAIL); this is necessary for the failure/abortion propagation scheme to work properly (see Section 4.4.2.6). Finally, as the guard spawning operation involves the creation of a guard tuple and goal records, it may take a relatively long time (when compared with the other operations). A new stage, SPAWNING, is added in the proposed model so that the completion of the spawning operation can be reflected by a stage changed: from spawning to spawned.

Appendix A3
Run-Time Structures in the Proposed Execution Model

This appendix demonstrates the creation and interconnection of the run-time structures in the proposed execution model. The following Concurrent Prolog program is used as an example:

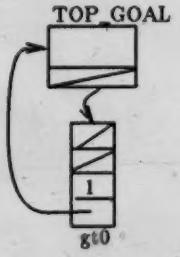
```
<- p(X),q(X?),r(Y).

[C1] p(1).
[C2] q(Y) <- u(Y) | t(Y).
[C3] q(Z) <- s(Z) | u(Z).
[C4] r(Z) <- s(Z) | .
[C5] s(1).
[C6] t(1).
[C7] u(1).
```

For the sake of simplicity, only the *Guard_Tuple* and *GT_Vector* fields of a goal record are shown with the latter placed inside the box representing the goal record, and the former along the boundary. During program execution, snapshots of the run-time structures will be displayed.

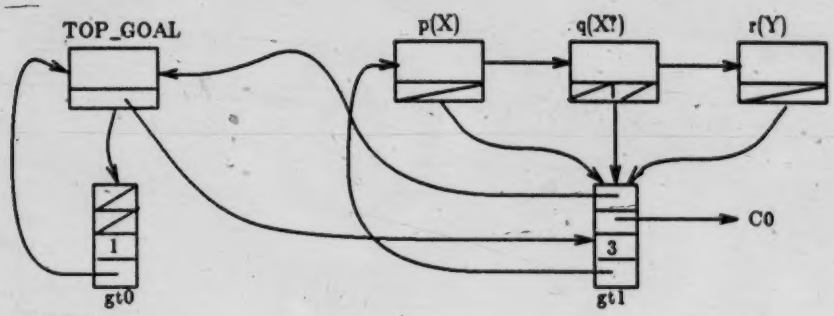
As explained in Section 4.4.1, before program execution, a top level goal TOP_GOAL is created whose only applicable clause C0 has a guard containing the goals in the original goal clause and an empty body:

```
<- TOP_GOAL.                          [new goal clause]
TOP_GOAL <- p(X),q(X?),r(Y) | .       [C0]
```

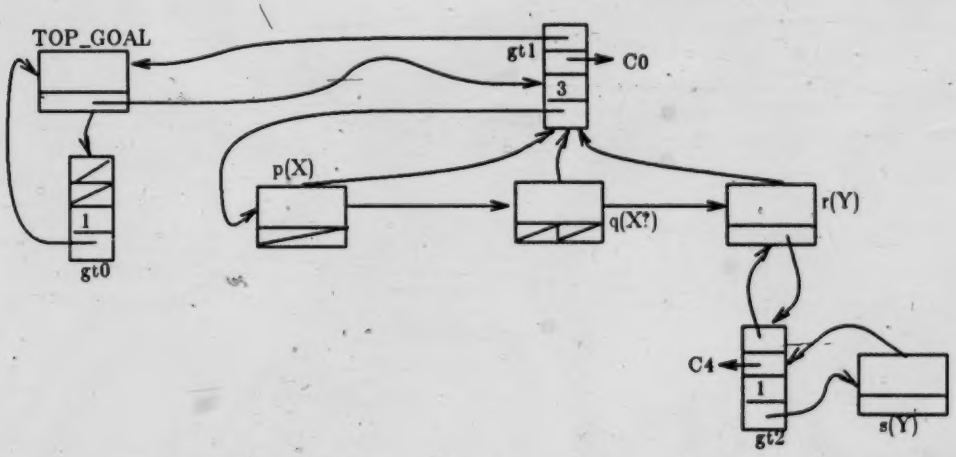
The system creates a goal record for TOP_GOAL. Since TOP_GOAL is the root of the AND/OR tree and hence not a part of any guard, a dummy guard tuple gt0 is created with NULL father and clause fields (used for program termination detection).



The system adds TOP_GOAL's goal record into the ready goal pool. Some time later it will be dequeued and processed. The clause C0 is applied and the head unification succeeds, the guard is spawned with a guard tuple gt1 created to monitor it. The goal records for p(X), q(X?) and r(Y) are created and added into the ready goal pool. The ready goal pool is now: {p(X), q(X?), r(Y)}.

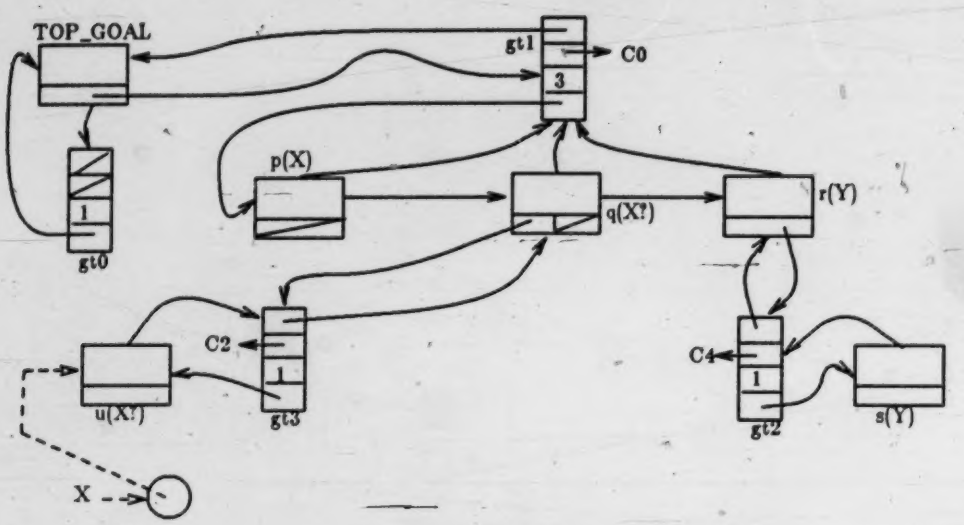


Suppose the goal record of r(Y) is dequeued and processed first. The head unification of r(Y) and C4 succeeds, binding Z to Y. The computation proceeds to spawn the guard in C4: a guard tuple gt2, and a goal record for s(Y) are created with the latter being added into the ready goal pool. The ready goal pool is now: {p(X), q(X?), s(Y)}.



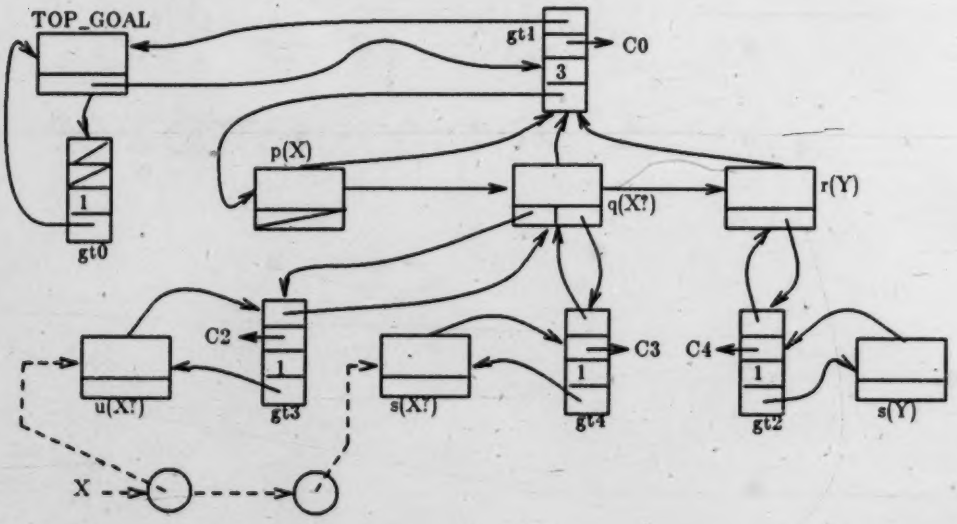
Suppose at this time the goal record of q(X?), which has two applicable clauses, is dequeued and processed. As the system tries the clauses *in sequence*, C2 is applied first. The head unification of q(X?) and C2 succeeds, binding Y to X?. The computation proceeds to spawn the guard in C2: a guard tuple gt3, and a goal record for u(X?) are created with the latter being added into the ready goal pool. The ready goal pool is now: {p(X), s(Y), u(X?)}.

Suppose the goal record of u(X?) is immediately dequeued by a process. The head unification of u(X?) and C7 requires the instantiation of the read-only variable X?, hence it is suspended. A suspension record (represented by a circle) referencing the goal record of u(X?) is created and added to the suspension list of variable X. The ready goal pool is now: {p(X), s(Y)}.

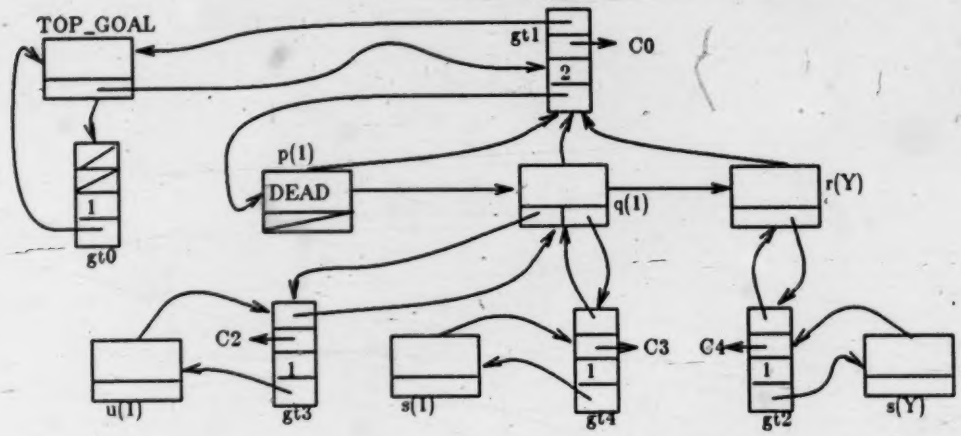


Meanwhile, the process evaluating q(X?), after having spawned C2, tries its second clause C3. The head unification of q(X?) and C3 succeeds, binding Z to X?. The guard is then spawned: a guard tuple gt4 and a goal record for s(X?) are created with the latter being added into the ready goal pool. The ready goal pool is now: {p(X), s(Y), s(X?)}.

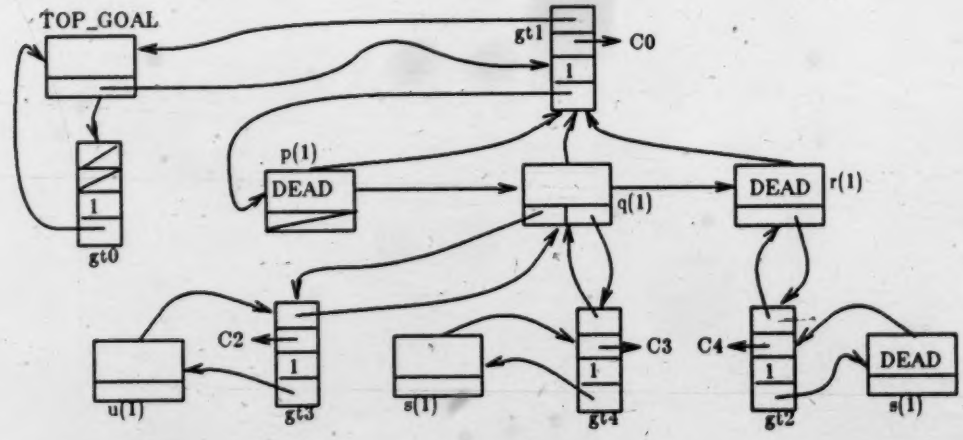
Suppose the goal record of s(X?) is immediately dequeued by a process. The head unification of s(X?) and C5 requires the instantiation of the read-only variable X?, hence it is also suspended. A suspension record is created for it and added into X's suspension list. The ready goal pool is now: {p(X), s(Y)}.



At this time the goal record of p(X) is dequeued and processed. The head unification of p(X) and C1 succeeds, binding X to 1. C1 has an empty guard, hence the computation proceeds immediately to COMMIT without creating any guard tuple. After the commitment action succeeds, the suspended goals u(1) and s(1) (note that X is now bound to 1) are added back into the ready goal pool. P(1) is reduced (marked as DEAD) to C1's empty body, creating no new goals. The guard count in gt1 is decremented. The ready goal pool is now: {s(Y), u(1), s(1)}.

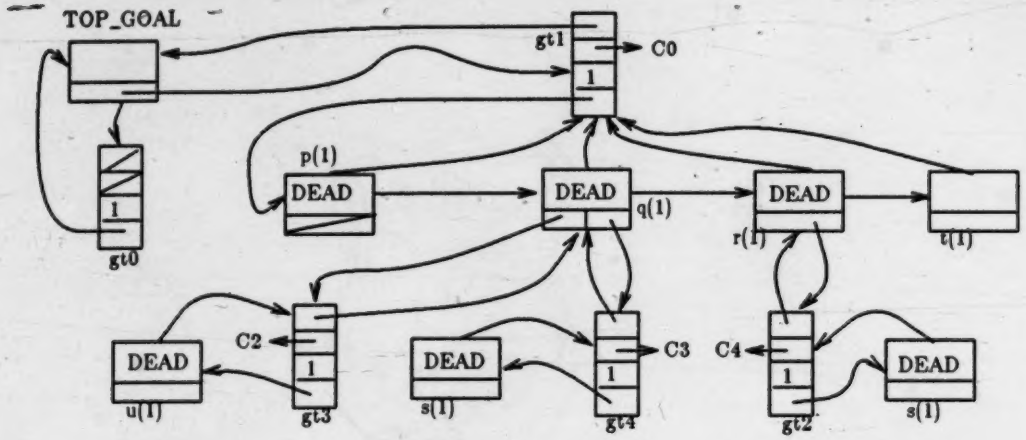


Suppose now the goal record of s(Y) is dequeued and processed. The head unification of s(Y) and C5 succeeds, binding Y to 1. Since C5 has an empty guard, s(1) commits immediately to C5 and is later solved (marked as DEAD). Also, the guard count in gt2 is 1, so a *recursive commitment* occurs: r(1) (i.e., s(1)'s father) is reactivated and commits to C4. As C4 has an empty body, r(1) is solved (marked as DEAD). The guard count of gt1 is decremented. The ready goal pool is now: {u(1), s(1)}.



Suppose now the goal record of u(1) is dequeued and processed. Eventually u(1) commits to C7 and is later solved (marked as DEAD). As the guard count in gt3 is 1,

q(1) is re-activated and commits to C2 (recursive commitment). C2's sibling clause C3 is KILLED, the goal s(1) in its spawned guard is also aborted (marked as DEAD). q(1) is reduced (and henced is marked as DEAD) to the committed clause body t(1). The goal record for t(1) is created and added into the ready goal pool and gt1's Goal_List. The ready goal pool is now: {s(1), t(1)}.



Suppose now the goal record of s(1) is dequeued. As its flag is not READY (since it is aborted after being inserted into the ready goal pool), it is discarded immediately. Finally the goal record of t(1) is dequeued and processed. t(1) commits to C6 and is later solved. As the guard count in gt1 is 1, TOP_GOAL is reactivated and commits to C0 (recursive commitment). As C0 has an empty body and the guard count in gt0 is 1, TOP_GOAL's father is referenced. Since it is NULL, the system knows that it is not a recursive commitment, but a *program termination*. The answers X=1, Y=1 are then reported.